INTO BASIC

SIEGMAR WITTIG

Translated by Siegmar Wittig et al
Edited by Henry Budgett


PET
commodore

INTRODUCTION

The BASIC tutorial course we are presenting over the next three months was specifically written for non-mathematicians, non-technicians and non-computer professionals. It is meant as a guide for amateurs, in the true sense of the word, who simply wish to program their own microcomputers.

With this aim in mind we have tried to keep the text free of technical details and have aimed at presenting the material in a comprehensible form rather than supplying information that is not strictly necessary. The book is written in the form of a tutorial course and, as such, should enable the reader to start producing BASIC programs fairly quickly. In order to get the best out of the book it is essential to work through the chapters in order, it would be difficult to do otherwise really as the book is in three parts!

The programs in the book are not based on any particular version of BASIC but take into account the various dialects offered on many of the popular home computers. All the listed programs have been tested on these computers and any variations necessary will be given in the text.

The various exercises given throughout the book are intended to test the skills you should have learned in each chapter. There are many ways to solve each of these but the 'preferred' method will be given at the end of the book — just so you can't cheat!

Design by MM Design and Print
Typesetting by Ebony Typesetting, Liskeard.
Printed by Alabaster Passmore & Sons Ltd, Maidstone.

Acknowledgements to German Translating Services and Tina Boylan

Contents

Part 1


ATARI
CBM

1. Thought Discipline

Algorithm — Programming Sequence

It is true to say that, on occasions, people are tempted to attribute an almost human intelligence to the computer. Using concepts and terms like 'Electronic Brain' or 'Computer Error' and 'the computer is fed with data' and 'it spits out the results' are no help toward understanding the way the computer functions. A computer cannot carry out any task which is not based on the plans of its programmer. By observing the programmer we are aware that the signs of intelligence which we thought we had discovered in the computer, are attributable to the skill of the programmer. Programming has become an activity that is important, interesting, fascinating and even relaxing for the owner of a home computer, and this activity lies within the wide field of both arts and crafts. To start with we will examine the aspects of programming most closely connected with the crafts. The programmer uses a number of tools, one of the most important being the program construction plan. This tool occupies a high place in the order of priorities. The term Program Construction Plan is often referred to by the word flowchart.

No matter which language is used in programming, the technicality of constructing a flowchart is always the same: It begins by the need to seek a method for the correct and complete solution of the problems. Among programmers this method of solution is also known as an 'algorithm'. Then a flowchart is drawn and is used as the basis for programming. In order to ensure that other programmers can understand the work carried out by their colleagues, certain conventions are observed when drawing flowcharts; see Fig. 1.

For the first example of drawing a flowchart we use the example of a young sports fan who, prior to joining a swimming club, enquires what his annual subscription rate will be.

What is the club's reply?

The annual subscription fee for a man is £10. If he is still a student, the annual subscription is half that amount.

The annual subscription for a woman is £8. Again, if she is a student she pays half that amount. Married women pay half the amount women students pay. In this example we don't need an algorithm to solve the problem of the young sportsman. Only a flowchart is needed to simplify the slightly confusing statement supplied by the club.

As an example imagine that you are a married woman. A glance at Fig. 2 would easily supply the answer to 'are you a man?' (no) and 'are you married?' (yes) as well as showing that the annual subscription would be £2. This example demonstrates the way in which a flowchart helps to simplify an apparently complicated statement of facts.

Exercise 1

Try to draw a flowchart about the information supplied by the club, varying from the one in Fig. 2 but leading to the same result. (Note that the Club does not specify a subscription for married women who are students.)

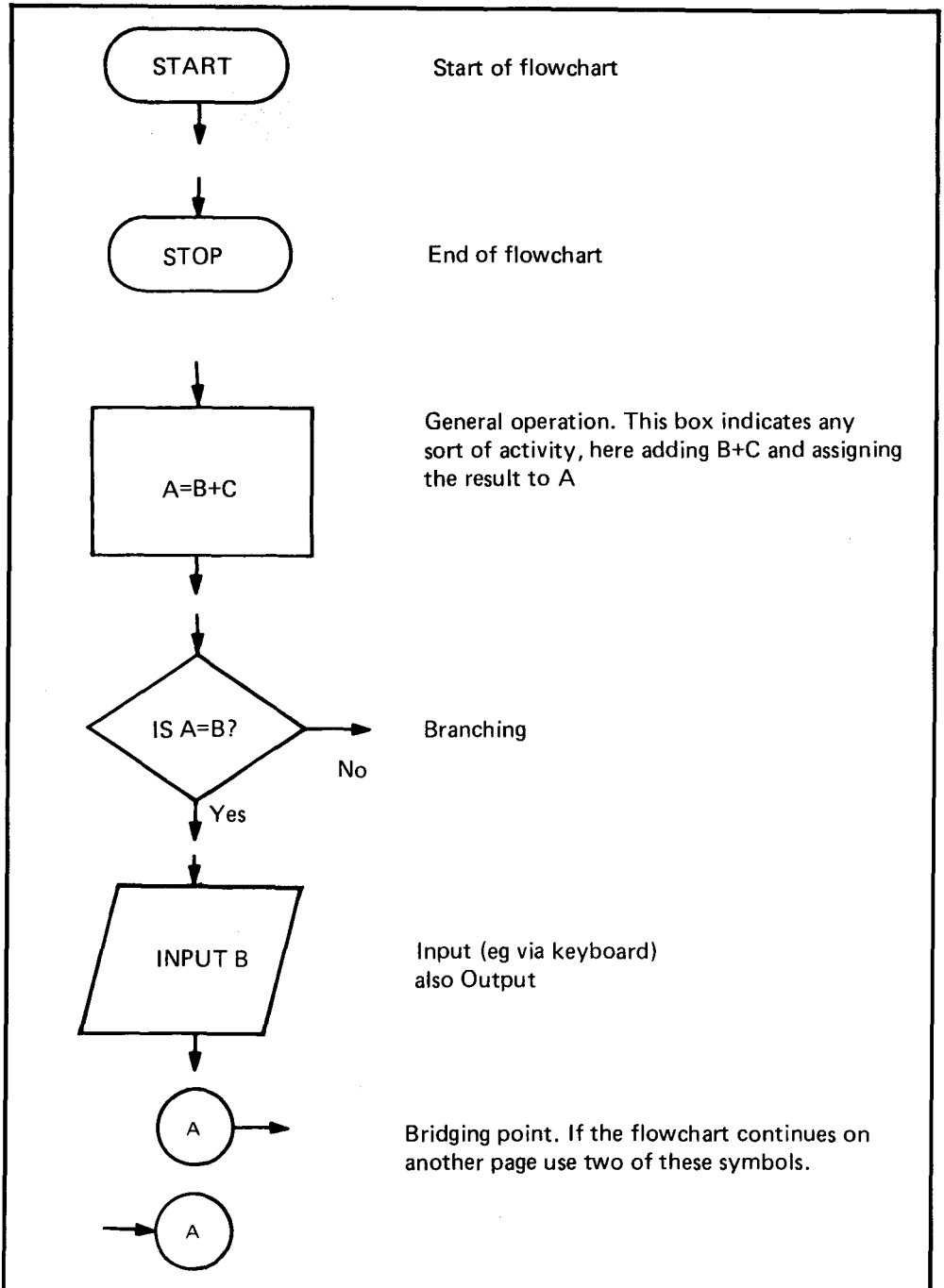


Figure 1. Some typical flowchart symbols

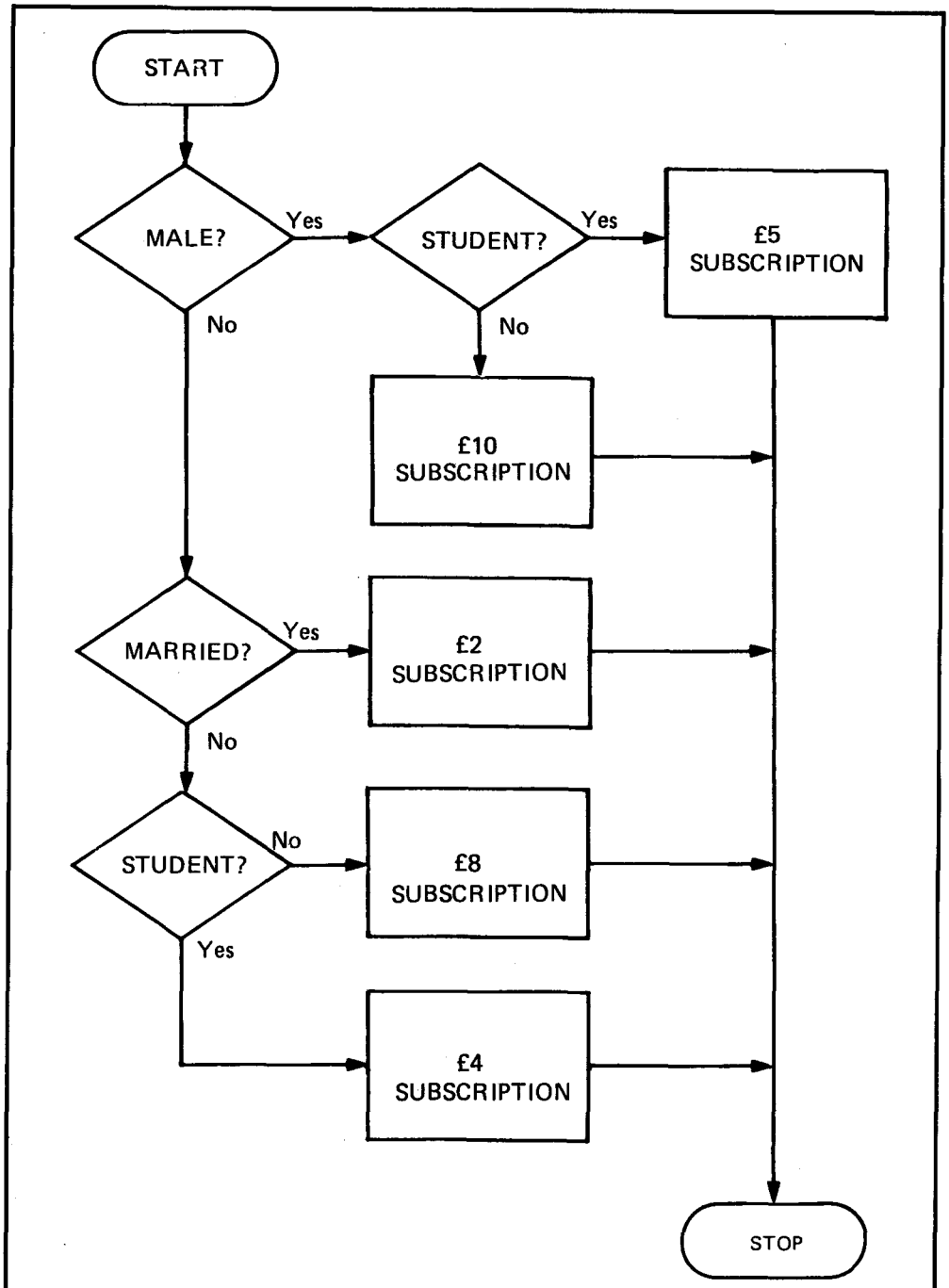


Figure 2. The sports club subscription flowchart


GRAPHICS
BLACK
DELETE
INPUT
POKE
PRINT
LOAD
LIST
LET
ENTER
NEXT
PAUSE
BREAK
SPACE
RUN
STOP
RETURN
SHIFT
HOME
CRSR

2. The First Steps

Symbols — Constants — Variables — Assignments —LET — PRINT — Flowchart — END — STOP — Command — NEW — RUN — CONT

It is rather strange to describe BASIC as a programming 'language', as this language is not actually spoken by those involved, neither by computer nor programmer. The communication between computer and programmer occurs in writing (but not using paper). On the other hand, the term 'programming script' would also sound strange. However, on closer inspection the use of the term 'programming language' seems reasonably justified: A natural language is, after all, expressed by symbols.

— By a certain vocabulary (there are three million words in the English language).
— By certain symbols representing the words (think of our own alphabet or Cyrillic, Greek or Arabic lettering).
— By a number of regulations which determine the composition of sentences, known as grammar.
— and, of course, by the meaning of words and sentences.

We also find these symbols in programming languages. All the recognised symbols of the programming language are referred to as characters in store; all the words used (and there are luckily only a few dozen of them) are the 'key words' and the rules about word combinations are usually called 'syntax' rather than grammar. As in natural languages there are also variations within programming languages. For example BASIC has many different dialects; almost every computer manufacturer has developed his own version of BASIC. There are, however, certain components which can be found in all the versions of BASIC. These will be the ones with which we will concern ourselves in this course.

Before we start programming, we will have to familiarise ourselves with certain terms in order to be able to understand the introduction to programming with BASIC.

Symbols

Everything in BASIC that we want to express in writing will be set out by means of certain symbols just as in any natural language. For obvious reasons the stock of symbols for BASIC was chosen as follows:

— 26 capital letters (A - Z)
— 10 numbers (0 - 9)
— so-called special characters
eg: ! " % ' () * = + / . , ; : $

Different BASIC versions can have a different choice of special characters. Often one finds such exotic ones as @ or # (hash).

Constants

A constant in BASIC is what is normally referred to as a number. Eg, 5 or 0.017 or 3.14 etc. In BASIC and data processing in general it is customary to differentiate between whole constants (so-called integer constants) meaning non-decimal figures like 7 or 3 or 1284) and decimal figures like 3.14159 or 0.72 or 0.001, (so-called real constants). These decimal figures are also referred to as figures with a fixed (fractional) point.

For decimals in BASIC there is yet another way of notation. Referring back to our mathematics lessons

5.1234×10^3 can be written instead of 5123.4

This is scientific notation or notation indicating to 'the given power of .....' (this type is also familiar from using pocket calculators). In BASIC, which lacks the means of depicting raised figures, we have to express these by writing 5.1234 E3. 'E' here means 'Exponent'. Written in this way the figure can also be fed into the computer.

Examples:

2.07×10^{12} is fed in as 2.07E–12
0.0209×10^{5} is fed in as 0.0209E–5 or .0209E–5
-12.3×10^{17} is fed in as 12.3E–17

Variables

In BASIC a variable is represented by a symbol or a group of symbols. Permitted symbols (names of variables) are the letters of the alphabet. Later we shall come across other examples of names of variables. Also among the variables we must differentiate between the several types. Here we want to start by working with simple variables. Such variables are already known in mathematics. One refers to a variable X or Y which means a size or value, and these can be altered during the course of the calculation. One can think of the variables in BASIC in similar terms: The name of the variable A is the symbolic sign for a storage space in your computer — storing a certain value. This value of the variable can be altered in the program while the program is being processed. It is still the same variable A, only with a different value. Besides the variable A, further variables (eg, L, P, X) can be used if they are needed to solve a problem.

The term variable usually presents the beginner with great difficulties. We will therefore return to this point in our section dealing with the LET instruction.

In most home computers the name of a variable consists of

— one of the letters A to Z
— a letter followed by a figure eg. B1, D0, Z9
— two letters eg AZ, MX, UG

Often one knows in advance that during the running of a program the variable will only have integer values. In this case the names can be marked by adding the character %: eg, A% or X1%. Such variables are called 'Integer Variables' (Integer meaning whole figure). You generally require less space for the storage of your integer value than for a 'real variable' of a value of 3.14 or 5.7E–2.

In Chapter 7 we will come across string variables ('string' meaning a chain of characters). Such variables are marked by adding a $ (dollar) sign. When choosing names for your variables refer to your computer handbook. Names of more than two letters are often allowed and there are further types of variables eg, those with double accuracy, like the TRS-80 Level II. Variables which in this system carry the symbol # as the last character in their name are stored and processed to an accuracy of 16 decimal places.

Instructions

We already know that there are rules governing BASIC symbols, keywords and syntax. Certain symbols, namely letters and some special characters, have been used to make up names for variables. Such names for variables apart from constants and keywords, with which we are not yet familiar, as well as further symbols and special characters can form sentences in BASIC; where only 'sentences' conforming with the syntactic rules of the BASIC language are permitted. In the programming language these sentences are called 'instructions' because the programmer feeds his commands, wishes, problems and information in the form of instructions into the computer. One of the programmer's working processes is to transcribe problems, which are presented in the form of algorithms, into a series of instructions which have to be formulated in a language which the computer can comprehend; eg, BASIC.

There exists a BASIC instruction which can be used when presenting the computer with an unknown variable while at the same time nominating its value.

LET

The easiest way in which to define a variable in BASIC is presented by the LET instruction. Here are three examples of LET instructions:

a) LET X = 3
b) LET Y = X–2
c) LET T = T+1

The LET instruction is usually described as assignment and is one of the previously mentioned keywords.

Which are the individual steps? Before taking a closer look, we would like to draw your attention to the similarities of the BASIC language and the English language. Example a) could be translated as follows: 'Computer, let X be equal to 31'. In evolving the language BASIC an attempt was made to achieve a similarity with English in order to make it easier to learn.

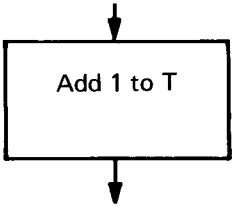
In connection with the LET instruction here are some extra comments: The equal sign in this assignment does not mean equal in the accepted sense of the word, but means 'is replaced by the value of'. Example (b) would have the following meaning: The value contained in the storage space representing the variable Y should be replaced by the value of the storage space representing the variable X decreased by 2. Admittedly, this sounds rather combersome, so in the language of the programmer this would be simply expressed as LET Y equal X–2.

Example c) would therefore mean:

LET T = T+1

A reader with mathematical training would obviously find this nonsense; indeed almost hair-raising. Well, this is easily clarified. If the storage space for variable T had the value of 2 before processing the instruction, the same storage space would have the value 3 after processing.

Try to give this process a lot of thought, it is of major importance. In the previous chapter we dealt with flowcharts, where in the flowchart the following box appears:



The following instructions would appear in the corresponding program

```
LET T = T+1
```

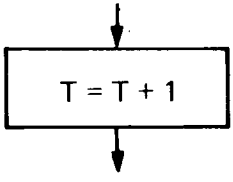
It must be pointed out that a LET instruction is carried out from right to left. The assignment on the right-hand side of the equal sign is calculated first, the result is then assigned to the variable on the left-hand side. For this reason there can never be more than a single variable on the left and complicated expressions must be placed on the right-hand side of the = sign.

It also follows from the way the computer interprets the LET instructions that the variables on the right-hand side of the = must already be known to the computer. Should the variable T in the LET instruction T = T+1 appear for the first time in the program, the right-hand side will not be calculated, since no value for T had been stored. Computers behave differently in these circumstances: they either report the programming error, or they use whatever value happens to be in the storage space for T, (where, we hope, this will be noticed in time) or they assume the value of 0.

We must mention here, that the word LET can be omitted in most BASIC versions. The two instructions

```
LET T = T+1 and
    T = T+1
```

have the same meaning. If we have the following box in the flowchart:



we have a valid BASIC instruction.

In order to practice with the LET instruction and also gain a better idea of the concept of variables, here is a small programming example.

Program example 1: Exchange the values of two variables

The value of the variables A and B should be exchanged. One could assume,

```
LET A=B
LET B=A
```

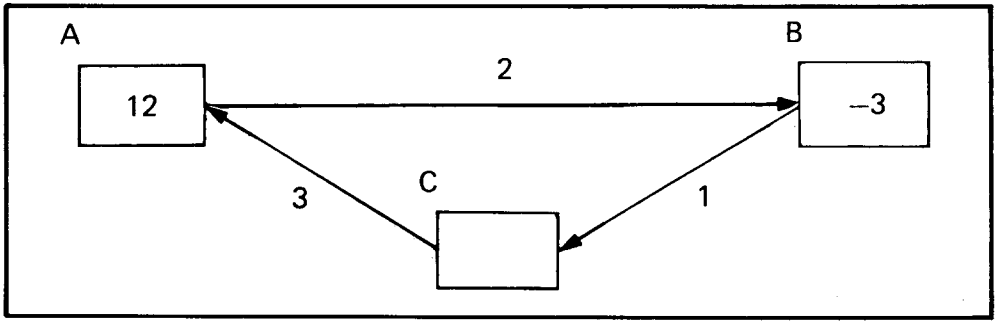
But you would, unfortunately, be wrong! Remember that we have to think of the variables A and B as symbolic descriptions of two storage spaces of some sort of value:

Let us take an everyday case as an example:

You want to carry out an oil change in your car and get the required amount in a tin can. You are confronted with the task of exchanging the contents of the engine block (namely old oil) with the contents of the can (new oil). In order to return the can filled with old oil to the garage there is only one possible route: You need an empty container into which to pour the old oil as an intermediate storage vessel. Then you can pour the new oil into the engine block. Finally you can pour the oil from the intermediate vessel into the can and the exchange is completed.

We can draw conclusions for our programming problem from these simple facts: We are no longer dealing with old and new oil, but with the contents of variables A and B. These contents should be exchanged. This can only be accomplished by using an intermediate vessel. Any new variable will serve this purpose, we shall call it C.

Storage Unit



Transfer of the contents from B into the storage cell C can be achieved with the command LET C = B

(Remember: The value from the right-hand side will be transferred to the left of the = sign.)

C then acquires the value -3. Comparisons have their drawbacks, and this applies in our case. Unlike the motor block, which was empty after the old oil had been drained, the variable B retains the previous value of –, or rather the assignment. LET C = B means that C has the same contents as B, or rather that the contents of B are copied into the storage cell with the name C. The second step is to also copy the contents from A (namely 12) into the storage cell B, which is achieved with:

```
LET B = A
```

The value previously attributed to B now becomes irrelevant since that value will be overprinted.

The third step makes the transfer of the contents of C (namely -3) into the storage space of A, which is done by

```
LET A = C
```

In this way the old contents of A (namely 12) are overprinted; after processing, the assignment -3 is in A. Result: A contains -3, B contains 12: the exchange has been completed. The variable C is no longer required, so we can forget it or use it somewhere else in the program for another purpose. The exchange program is as follows:

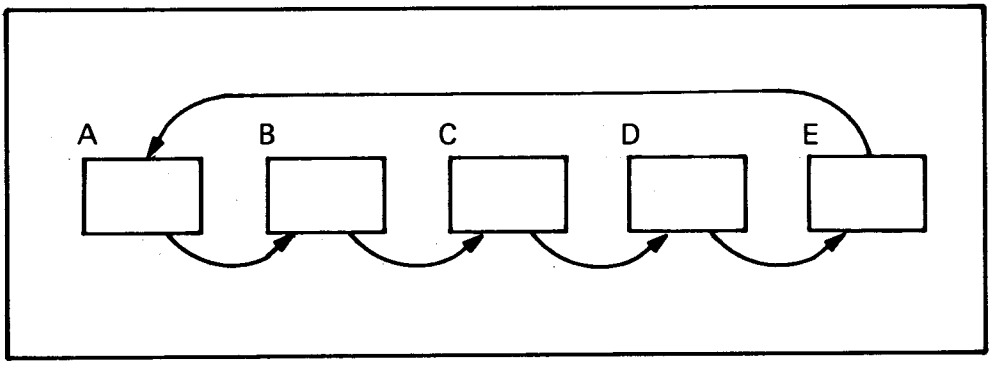
```
LET C = B
LET B = A
LET A = C
```

Maybe somebody doing an oil change will get the idea of pouring the new oil into a tub in order to drain the old oil into the can and then pour the new oil from the tub into the car. This would be another way to accomplish the exchange of old for new. Our program would work accordingly:

```
LET C = A
LET A = B
LET B = C
```

The result is the same, and that's what counts.

Program Example 2: Exchange the values of several variables



The contents of the variables should move to the next space and the contents of the last variable E should reach A.

Solution:

```
LET H = E
LET E = D
LET D = C
LET C = B
LET B = A
LET A = H
```

This is only one of several possible solutions. Instead of E at the beginning of the program, one could store any variable in the intermediate storage space, which we called H. The choice of names is of course optional, only no name can be used twice. Hopefully you now find that BASIC assignment is not all that mysterious.

PRINT

So far we have either allocated BASIC variable values or changed them around. We have not yet received a single value from the computer as a way of seeing the results of our efforts on the screen. We will now make up for that and so will concentrate on BASIC's most important process.

The output instruction of BASIC is the PRINT instruction. Here is a simple example:

```
Y = 1937
PRINT Y
```

The first line is not really connected with the PRINT instruction, it merely serves as a definition of the variable Y and the allocation of a value. A number of such sentences will be called a program. From now on the output of this program looks as follows:

```
19.37
```

What you saw in the program above was the PRINT instruction in its simple form. There are a great number of variants of this instruction. Let us assume we want to print out one line with two values, where the value of a variable X will be in the first place and that of variable Y in the second place.

The program would look like this:

```
X = 19.37
Y = 201.6
PRINT X, Y
```

The output of the program would look as follows:

```
19.37          201.6
```

The computer can subdivide each output line on the screen (or printer) into four or five

parts. The number and the length of these sub-divisions depends on the computer and the length of the line it can produce. Each sub-division is called a 'print zone'. Whenever a new line is started, the first output number always appears in the first print zone. If, as is the case in our program, a comma appears in the print instruction the computer realises that it has to print more data on the same line; the computer moves the 'cursor' (the sign indicating the next position on the screen) to the start of the next unused print zone. If your computer, for instance, has print zones ten spaces long and the output in the first zone is 12 symbols long and, therefore, runs into the second zone, the computer prints the next output in zone 3, because zone 2 has already been partly used up. (Take note that the blank spaces in between the words also count!)

With PRINT instructions one can not only get an output for numeric values but also for text. For examnple one can get the computer to write the text 'TODAY IS WEDNESDAY' on the screen. The PRINT instruction required is as follows:

```
PRINT "TODAY IS WEDNESDAY"
```

You can see from this example that the text of a PRINT instruction is always in inverted commas ("...."). The inverted commas tell the computer that the text between them should be printed out in the same form. The only symbols that can never appear in a text are the inverted commas themselves. Should you, however, require this symbol in a text, then use the apostrophe:

```
PRINT "TODAY IS 'WEDNESDAY' "
```

appears as

```
TODAY IS 'WEDNESDAY'
```

Take note of the following example

```
X = 6
PRINT "THE VALUE OF 'X' IS ",X
```

The output line looks like this:

```
THE VALUE OF 'X' IS          6
```

Note that gap between the text and the number 6. This is because the text takes up 20 spaces and the computer print zone is 10 spaces. The first print zone has been filled and part of the second zone has also been taken up. The value of 6 appears in the third print zone. We, of course, would prefer the printout in the following form:

```
THE VALUE OF 'X' IS 6
```

On most computers this can be achieved by substituting a semi-colon for the comma

```
PRINT "THE VALUE OF 'X' IS ";X
```

The semi-colon has the same general effect as the comma in that it tells the computer that there is more to follow in the same line. It differs from the comma in so far as it does not separate the print zones, but instructs the computer to print the whole text continuously,

disregarding print zones. According to the type of computer there may be gaps of 0, 1 or 2 spaces in the printouts.

Program Instructions

You have now worked your way through many pages dealing with the theory without having touched a computer. This situation will be remedied without delay! First of all we have to learn a few important details which we should observe when handling a computer. We have already mentioned that a program consists of a series of instructions. In BASIC every instruction has to start with a line number. This is a positive integer number which must not exceed a certain size. The size depends on the type of computer. If you have fed several instructions into the computer and taken care that each instruction number appears only once, the instruction will be carried out by the computer, starting with the lowest number. You could feed in:

```
10  X = 2                or    30 PRINT X,Y
20  Y = X + 3                  20  Y = X + 3
30  PRINT X,Y                  10  X = 2
```

In both cases the computer processes line 10 first. You can see that you do not have to use numbers 1, 2, 3. On the contrary, it is wiser to leave gaps in the succession of numbers in order to have the opportunity to intersperse extra lines (when extending the program).

What you should not do is the following:

```
30PR INTX,Y
```

If you put in blanks in this or any other BASIC keywords, the computer will, as a rule, give up. (Blank is the term for an empty space.) But here too you will find differences among the various computers.

END

The last line of a program should contain the instruction END.

With most computers this is of minor importance, but may be relevant in certain flow-charts.

```
10 X = 2
20 Y = X + 3
30 PRINT X, Y
40 END
```

STOP

Often it can be advisable to interrupt a program before getting to the end so one uses STOP. STOP used in, for example, line 120 shows up on the screen after completion of the program with the announcement BREAK IN 120. STOP can be used several times within one program, this also applies to END. If one wants to interrupt a program for test purposes STOP is preferable. A method by which one can continue a program after an interruption is given in the section about CONT.

Commands

When you have fed in the program as explained in the previous section, the computer will, in spite of the allocated number, fail to deliver the expected result. Another detail should be noted.

Apart from the program instructions there is a further group of orders, which we will refer to as commands. Commands are often absolutely nothing to do with the program itself; rather they serve generally in the course of the development of the program. We will introduce two commands.

NEW

NEW erases all stored program lines and prepares the computer for a new program.

RUN

With RUN the processing of the program is started, it begins at the lowest line number. These commands, which are also referred to as 'direct' commands, are fed in without line numbers. The command is carried out immediately and is not stored as part of the program.

Your first program could be fed in as follows:

```
NEW          (Then press the Return, Enter or Carriage Return key)
10 X = 2     (as above)
20 Y = X + 3 (as above)
30 PRINT X Y (as above)
40 END       (as above)
```

Now we have stored the program in the computer. By typing the command RUN (followed by depressing the Return key) you command the computer to actually proceed with the program. The result appears

```
2             5
```

on a new line on the screen.

CONT

After an interruption of the program caused by STOP one can give the command CONT (continue). The processing of the program now continues until either a further STOP, an END, or the last instruction is reached. If the program was interrupted because of a programming fault, one cannot proceed with the help of CONT; the mistake has to be corrected.

Subscriptions

Are you looking for a more personal approach to computing? You are . . . then Computing Today is the magazine for you! Packed full of feature articles, projects, general topics, news and reviews, Computing Today is aimed at readers who want to get more out of their microcomputer.

The latest ABC circulation figures show Computing Today has increased its readership by 85% over the previous year — great news for us at CT. However, the ever increasing demand for Computing Today has meant that, despite our printing more each month, some readers seem to be missing out on their regular copy.

If you would like to ensure a regular supply for the next twelve months, each issue lovingly wrapped and posted to you, nothing could be simpler. Just fill in the form below, cut it out and send it with your cheque or Postal Order (made payable to ASP Ltd) to:

Computing Today Subscriptions,
513 London Road,
Thornton Heath,
Surrey CR4 6AR.

Alternatively you can pay by Access or Barclaycard in which case simply fill in your card number, sign the form and send it off. Do NOT send your card.

Do yourself a favour, make 1982 the year you start to take Computing Today every month and we'll give you a truly

Personal Approach To Microcomputing.

SUBSCRIPTION ORDER FORM

Cut out and SEND TO :
Computing Today Subscriptions
513, LONDON ROAD,
THORNTON HEATH,
SURREY,
ENGLAND.

Please commence my personal subscription to Computing Today with the ..... issue.

SUBSCRIPTION RATES
(tick ☐ as appropriate)

£12.10 for 12 issues UK	☐
£15.75 for 12 issues Overseas Surface	☐
£35.35 for 12 issues Overseas Air Mail	☐

I am enclosing my (delete as necessary)
Cheque/Postal Order/International Money Order for £.....
(made payable to ASP Ltd)
OR
*Debit my Access/Barclaycard**
*(*delete as necessary)*

Please use BLOCK CAPITALS and include post codes.

Name (Mr/Mrs/Miss) ..... (delete accordingly)
Address .....
.....
.....
Signature .....
Date .....

3. The Calculation

Arithmetic Operators — Expressions — Assignments

As BASIC took shape, its initiators took pains not only to link the BASIC commands with the English language, but also to form the problems to be solved by the computer in such a manner that the procedures and notations encountered in normal maths were incorporated. An elementary calculation, or 'operation', such as addition, subtraction, or multiplication is mathematically expressed using the operators + (plus), – (minus) and * (times). These operators are also found in BASIC.

Arithmetic Operators

In every version of BASIC there are at least five arithmetic operators:

Operator	Meaning
=	. . . is replaced by the value of . . .
+	addition
–	subtraction
*	multiplication
/	division

Many computers have a sixth arithmetic operator, ↑ , meaning 'raise to the power of. . . . Other symbols, instead of ↑ , such as ** or ∧ are also employed.

At this point, a few comments are necessary: The equals symbol (=) does not mean 'equal to', as in mathematics, but triggers a step in the computer, whereby the value of the side to the right of the equals symbol is allocated, or 'assigned' to the variable on the left-hand side of the equals symbol. This has already been described in the LET statement. The equals symbol is a rather poor choice for the description of this procedure. The notation A←2 instead of A=2 would provide much more clarity. Indeed, other programming languages exist which use the symbol ← to describe an assignment.

As expression of the form

6 = 2 + 4

which is valid and commonly used in mathematics, is absolutely meaningless in BASIC.

Before we progress further, a program example is given which you can feed into your computer.

Program Example 3: Arithmetic Operators

```
10 LET A=3+4
20 LET B=5-6
30 LET C=3*15
40 LET D=6/3
50 LET E=2↑3 or   50 LET E=2**3   or   50 LET E=2∧3
60 LET F=A*E
70 PRINT A,B,C,D,E,F   or   70 PRINT A;B;C;D;E;F
```

The result should read: 7 -1 45 2 8 56

The BASIC notation for exponents will at first appear unusual:

```
50 LET E=2 ↑ 3      for E=2³
```

If the two forms of the statement

```
50 LET E=2 ↑ 3      or      50 LET E=2**3
```

do not have an effect on your computer, they try it with

```
50 LET E=2*2*2
```

in which case your computer only has a simple BASIC version and cannot deal with exponents.

Using the statement 60 LET F=A*E we can indicate the different usage of the arithmetic operators in BASIC, or mathematics respectively. In mathematics, the multiplication symbol is usually omitted.

$C = 2\pi r$ instead of $2*\pi*r$

or

$a^2 - b^2 = (a + b)(a - b)$ instead of $(a + b)*(a - b)$

The multiplication operator must never be omitted in BASIC:

```
LET F=A*E
```

The statement 60 LET F=AE would cause the computer either to issue an error message or to proceed in a different way from that planned.

Expressions

A programmer uses the term 'expression' to denote formulas containing constants, variables, and operators. A more formal description would be: An expression is a combination of variables and/or constants (numbers), which are linked together by one or more arithmetic operators. The formulas 3*X+4, A-2, and A*(A+B)/(2-C) are all examples of expressions.

Assignments

If an expression is allocated to a variable by an equals symbol, then one speaks of an 'assignment'. Thus:

```
Y=3*X+4
B=A-2
Q=A*(A+B)/(2-C)
```

are all examples of assignments.

The processing of assignments has a few special features which are worth further examination.

Suppose the variable X has a value of 2. Given the assignment

```
Y=3+X*X
```

what is the numerical value of Y? If your answer is anything other than 7 or 10, then you need to polish up your arithmetic. For those who came up with 7 then the value 10 may come as a surprise. It all depends on whether you used your pocket calculator or computer. The calculator would have used the following system:

```
Y=(3+X)*X     or     Y=(3+2)*2
```

The operations are carried out in the same sequence in which they appear. The result is 10.

A computer would have reckoned differently:

```
Y=3+(X*X)     or     Y=3+(2*2)
```

giving a result of 7. This may appear curious, and is due to the fact that computers carry out arithmetic operations in a definite sequence. For that reason, one speaks of an 'operator hierarchy'. The computer examines the expression from left to right and carries out all multiplications and divisions. Then it goes through again, carrying out additions and subtractions. The only possible way to manipulate the sequence of operations is in the placing of brackets. If a computer registers a group of brackets, or, as we say, nested brackets, then the calculation begins with the innermost bracket pair.

The result of the expression

```
3+(2*(3+1*3))/(2+1)
```

is 7. The innermost bracket pair contains 3+1*3, giving a value of 6. In the outer bracket pair, 6 is multiplied by 2, giving 12. Moving on to the last pair of brackets, a value of 2+5=3 is obtained. All the brackets have now been considered, and the expression takes on the form

```
3+12/3
```

Division is carried out before addition, thus

```
3+(12/3)     or     3+4
```

is the solution. The result is 7.

Try to calculate the following expression:

```
7+((7*8)/2)/(((12+8)*2)/20)
```

(The result should be 21.)

Having done that, try to remove all unnecessary brackets and operators in the expression without altering its value. The solution should read:

```
7+7*8/2((12+7)*2/20)
```

Here is an example for use with the computer.

Program Example 4: Expressions

```
10 LET A=-10
20 LET B=A/2
30 LET C=((A-1)*2+B)*3
40 PRINT A,B,C
```

The answer should be: -10 -5 -81

Exercise 2

Try to write a program that calculates the following expressions (for a=2, b=3, c=4).

Expression	Result
$a^2+2ab+b^2$	25
$(a+b)(a-b)$	-5
$a^2c+2abc+b^2c$	100
$(1+a(b+c))/c$	5
$((2a+b)/(a+b-1))/(c-a)$	.875

4. How a Computer Reads

INPUT — REM — LIST — Program Alterations

With BASIC it is possible to interrupt a running computer, ie one which is carrying out the program, at any point enabling the user to feed in data. The BASIC keyword was chosen accordingly: INPUT.

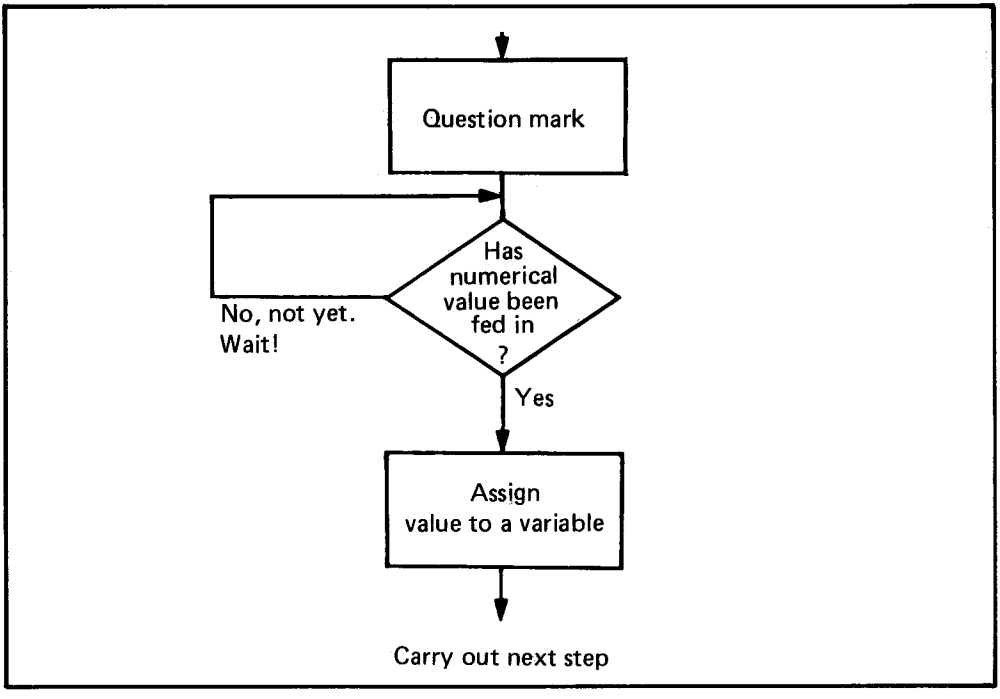
INPUT

Using the INPUT statement the user can insert a number into the program from the computer keyboard. Suppose you want to write a program which calculates the square of a number. The program could be something like:

```
10 INPUT A
20 PRINT A*A
```

The lines are treated individually. Line 10 has the following effect: As soon as the computer realizes that an INPUT statement is involved, it makes clear to the user that it needs data. Some computers do this by printing a question mark (?) on the screen. The computer now goes on stand-by. Now it's up to the user to enter the value that the computer can assign to the variable (in this case A). When this step has been completed, the computer goes on to the next line. In our example line 20 prints the numerical result of A^2 on the screen.

The INPUT statement can be seen as a shortened form of a procedure which one could describe in the following way:



The user always has to depress the Carriage Return (Return, Enter) key to complete the entry. Only then can a number be assigned to a variable and the program continued.

The INPUT statement possesses many variants, eg:

```
10 INPUT A,B
```

This statement causes the question mark to be printed and then waits for the entry of the numerical value of A, after which a second question mark is printed, and again it waits, this time for the value of B.

Another possibility is that both values, separated by a comma, are entered after only one question mark has appeared.

The BASIC system is acting as a prompt in printing the name of the variable to which a number is to be assigned. The programmer can contribute an improvement by formulating a specific question or request, which allows a closer description of the input. This is illustrated in the following example:

Program Example 5: Multiplication Program

```
NEW

20 PRINT"ENTER TWO NUMBERS FOR MULTIPLICATION"
30 INPUT A,B
40 PRINT"THE RESULT IS:"
50 PRINT A;"*";B;"=";A*B

RUN
```

For example, enter the numbers 7 and 9, then the computer should calculate:

```
7*9=63
```

Preceding the necessary INPUT pause, the program prints the request

```
ENTER TWO NUMBERS FOR MULTIPLICATION
```

This precisely describes the arithmetic operation which the data will be subjected to. Prompting in this manner is absolutely essential in the case of long programs or when the user is not familiar with the program. A further simplification of the INPUT statement is also possible on some systems. The steps

```
20 PRINT"ENTER TWO NUMBERS FOR MULTIPLICATION"
30 INPUT A,B
```

can be replaced by

```
30 INPUT"ENTER TWO NUMBERS FOR MULTIPLICATION:";A,B
```

Thus (and this may at first be confusing) the INPUT statement serves not only to accept data, but also to print out text.

REM

Now that we have worked our way through the necessary fundamentals, our knowledge of the BASIC language will grow quickly. The programs will become increasingly longer, and you don't need to be ashamed if you have forgotten what you programmed a few months ago, or even yesterday for that matter. Every programmer knows this feeling, and the BASIC language is conveniently equipped with a statement allowing comments to be introduced. The keyword in this case is REM and stems from the word remark. A program including the REM statement could look like:

```
10 REM MULTIPLICATION PROGRAM
20 INPUT A,B
30 PRINT A*B
```

The keyword REM must appear at the beginning of every commentary line. Apart from that, you are allowed a free hand in the formulation and number of comments. The computer stores the comments, but they don't play any active role in the program. If you want your program displayed on the screen (we'll see how a little later) for reasons of supplementation or alteration, then the comments appear as well. You should get used to prefixing important and difficult program steps with a REM statement, it will help the legibility of the program. The only drawback is that the REM statement still occupies memory, and this has to be kept in mind if the available storage is getting a little short.

LIST and Program Alterations

As programs become longer, the wish to alter them grows. It would, therefore, be practical to be able to have the stored program displayed for 'trouble-shooting' and making alterations. One of the commands that, like NEW or RUN, is instantly carried out by BASIC after entry is LIST. This causes the program to appear on the screen.

The LIST command is very versatile. One can not only have the complete program listed, but also choose between parts of the program and even individual lines.

```
LIST 30
```

for example causes the computer to display line 30 on the screen (provided a line 30 exists). LIST 20-70 lists all the lines between 20 and 70 inclusive. Another form of this command is

```
LIST 20,70
```

The command

```
LIST 70- (or LIST 70,  )
```

would be accepted by many computers. This has the effect of listing a program from line 70 (or, if line 70 does not exist, then from the next line) to the end of the program.

Conversely

```
LIST -70 (or LIST ,70  )
```

means that the program is listed on the screen from the beginning to line 70 inclusive.

Here is an example of the different uses of LIST which you can try out on your computer. E indicates your entry.

```
E: NEW
E: 10 LET A=2
E: 20 LET B=3
E: 30 PRINT A,B,A+B
E: 40 END
E: RUN

         2         3         5

E: LIST
   10 LET A=2
   20 LET B=3
   30 PRINT A,B,A+B
   40 END

E: LIST 20
   20 LET B=3

E: 20 LET B=7
E: RUN

         2         7         9

E: LIST 20-40
   20 LET B=7
   30 PRINT A,B,A+B
   40 END
```

If you have carefully followed the different steps, you would have noticed that we listed line 20 and made a slight alteration to it before entering it again:

```
20 LET B=7
```

The entry of a line prefixed with a number which is already present in a program causes the old line to be erased. Thus one can alter programs in quite a simple manner. If a program is to be supplemented then one only has to enter the new lines. This is shown in the following example:

```
E: LIST
   10 LET A=2
   20 LET B=7
   30 PRINT A,B,A+B
   40 END

E: 35 PRINT A-B

E: LIST
   10 LET A=2
   20 LET B=7
```

```
30 PRINT A,B,A+B
35 PRINT A-B
40 END

E: RUN
        2       7       9
        -5
```

If you need to remove a line in the program, you only have to enter the line number and subsequently depress the Carriage Return (Return, Enter) key. To remove line 35 from the program:

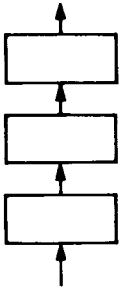
```
E: 35
E: LIST
   10 LET A=2
   20 LET B=7
   30 PRINT A,B,A+B
   40 END
```


SHARP

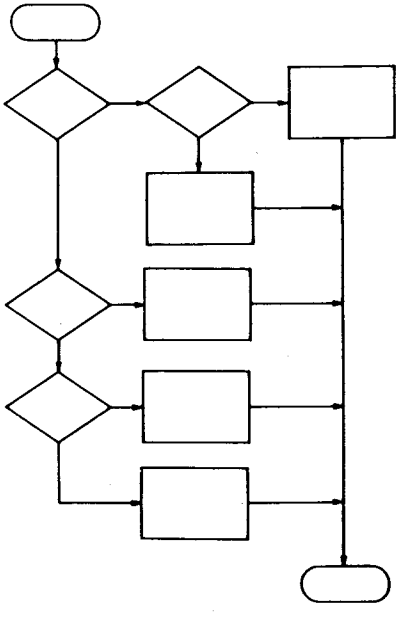
5. How to Divert a Computer from the Straight and Narrow

GOTO — IF. . .THEN. . . — Relational Operators

We have already seen that the BASIC Interpreter (this is the part of the computer which translates the BASIC program into a form or 'language' that can be understood by the computer) arranges the line numbers of the instructions into an ascending series and also processes the instructions in this sequence. A program on these lines, expressed as a flow-chart, would operate in the following manner:



However, we can recall flowcharts which differ quite considerably from this scheme. Diagram 2, for example, had the form:



In this case the 'linear' program structure has been dispensed with. The nature of our problems often require the programs to be flexible enough to allow deviations from the 'straight and narrow', for jumps back to a previous step, the repetition of statements, and even for the same program to be completed in more than one way. BASIC has controlling statements for just these cases. We will now have a look at a simple statement of this kind.

GOTO

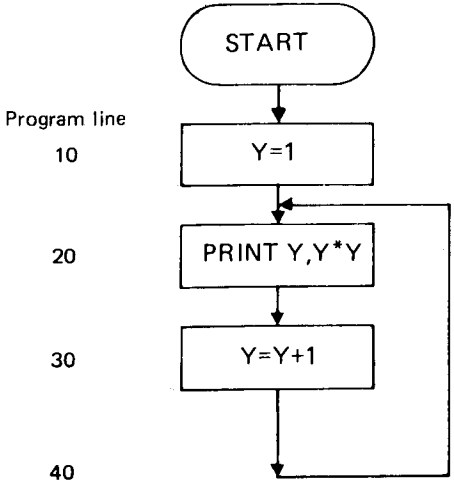
The simplest control statement in BASIC is GOTO. An example of its application can be shown as follows:

```
10 LET Y=1
20 PRINT Y,Y*Y
30 LET Y=Y+1
40 GOTO 20
```

The structure and meaning of the GOTO statement is quite simple. The keyword GOTO is followed by the line number of the statement which the computer is to carry out next. Thus the lines of our program example would be carried out in the following sequence:

```
10  20  30  40
20  30  40
20  30  40
20  30  40 etc.
```

The flowchart has the form:



The individual boxes represent the lines 10, 20 and 30, the GOTO statement isn't enclosed in a box. This statement is represented solely by the connecting line which leads back to a junction point between the line 10 and 20. For this reason the GOTO statement is known as a 'jump statement'. Let's see what the program actually does.

Line 10 is carried out first. The variable Y is assigned the value 1. The next line is concerned with printing the numerical value of the square of Y: 20 PRINT Y,Y*Y. If we take the instantaneous value of Y and insert it in this line, then we have, strictly speaking:

```
PRINT 1,1*1
```

Now we progress to line 30. Here the value of the variable Y is to be increased by 1, Y now has a value of 2. Line 40 is the GOTO statement, this tells us that the step to be carried out next has the line number 20. We therefore go back and write down:

```
20 PRINT Y,Y*Y
```

Inserting the instantaneous value of 2 for Y the line takes on the form:

```
20 PRINT 2,2*2
```

giving values of 2 and 4. This procedure will be repeated for Y=3, Y=4, Y=5 etc. The only snag is that there is no terminating step in the program or any possibility of jumping out of the 'loop' (this is a repeated part of a program). The instruction which makes this possible is one that we will shortly get to know.

From the lengthy explanations most of you will have realised that corresponding paired values for the function $Y=X^2$ are being printed. Whole numbers and their squares appear on the screen:

```
1     1
2     4
3     9
4     16
.     .
.     .
.     .
```

At present our short program represents an endless loop; ie, the same steps are being continually repeated with increasing values of Y. It is, therefore, desirable to have a method whereby the computer can be informed of the number of cycles that will complete a given loop. This is something that we'll be concerned with in Chapter 6.

IF. . .THEN. . .

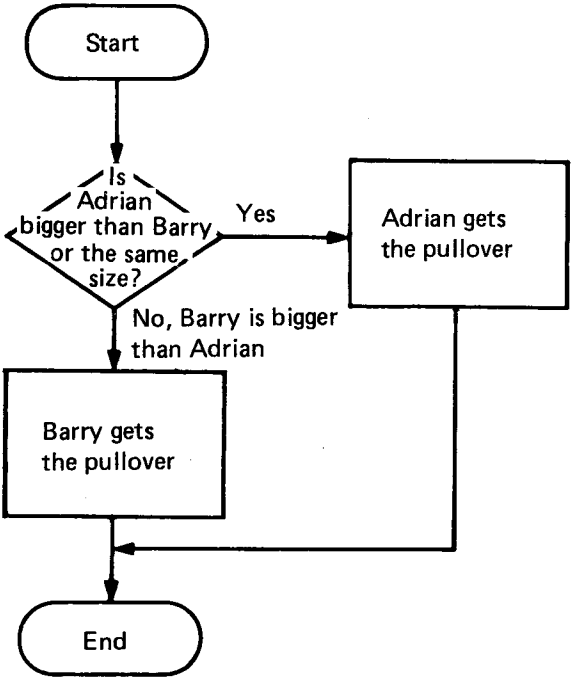
In a superficial assessment the question of the real advantage of the computer over the pocket calculator (that can be purchased considerably cheaper) could be raised. Later on we'll recognise where the noticeable strengths of the computer lie. One of its most important features is the ability to 'compare'. BASIC has a special statement for programming comparisons or conditions. Here, however, a preliminary introduction is necessary.

If we compare in normal everyday life it's because we want to make a statement concerning the result of the comparison, eg; 'Barry is bigger than Adrian' or 'the sum of my takings is less than the sum of my expenses'. All in all there are six conceivable different conditional statements, 'greater than' and 'less than' are two of them. In the following table

the conditional statements are on the left and the mathematical symbols used to represent them are on the right-hand side. These symbols, called 'conditional' or 'relational' operators, are also employed in BASIC.

Condition	Conditional Operator
. . .is less than. . .	. . . < . . .
. . .is greater than. . .	. . . > . . .
. . .is equal to. . .	. . .=. . .
. . .is smaller or equal to. . .	. . . < =. . .
. . .is greater or equal to. . .	. . . > =. . .
. . .is not equal to. . .	. . .< > . . . (sometimes written ≠ or ><)

How useful are these comparisons to us? More often than not they simply serve in establishing certain facts: 'Barry is bigger than Adrian'. However, sometimes a course of action depends on the result of a comparison. If Barry is bigger than his brother Adrian, then granny will knit him a new pullover (Adrian will grow into the pullover anyway, and can have it later). If Adrian is bigger than Barry, then Barry will get the pullover. If they are both the same size then Adrian will still get the pullover, because he didn't forget granny's birthday. This complicated family matter can be shown in a flowchart:



As we can see, there is a result in both of the possibilities.

We should now try to apply our knowledge of comparisons and their conclusions to the

ZX COMPUTING

Britain's Biggest Magazine Fo

Make sure you get every issue of *ZX Computing* — Now bi-monthly!

Just £11.50 will ensure the next six issues will be lovingly wrapped and posted to you. Just fill in the form below, cut it out and send it with your cheque or postal order (made payable to ASP Ltd) to:-

ZX Computing Subscriptions
513 London Road,
Thornton Heath,
Surrey CR4 6AR

Alternatively you can pay by Access or Barclaycard in which case simply fill in your card number, sign the form and send it off. Do NOT send your card!

Make the most of your ZX computer with *ZX Computing* — Now bi-monthly!

Subscription Order Form

Cut out and SEND TO:

ZX COMPUTING Subscriptions
513, London Road,
Thornton Heath,
Surrey CR4 6AR

Please commence my subscription to ZX Computing with the very next issue.

SUBSCRIPTION RATES

(tick □ as appropriate)

£11.50 for six issues UK □
£13 for six issues overseas surface mail. □
£23.80 for six issues overseas airmail. □

I am enclosing my (delete as necessary) Cheque/Postal Order/International Money Order for £.....
(made payable to ASP Ltd)
OR
Debit my Access/Barclaycard (*delete as necessary)*

BARCLAYCARD VISA

Please use BLOCK CAPITALS and include post codes.

Name(Mr/Mrs/Miss) .....
delete accordingly

Address .....

.....

.....

Signature .....

Date .....

Personal SOFTWARE

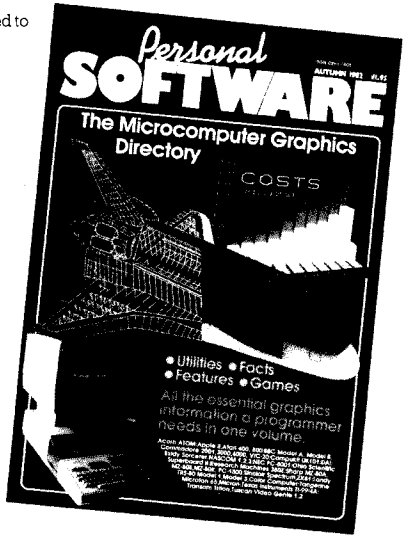
Personal Software is a quarterly publication dedicated to all aspects of software for the most popular micros.

Forthcoming issues will contain everything you ever wanted to know about machine code, utility programs and simple software techniques.

Personal Software can be ordered directly from us at £7.80 per annum or £1.95 per copy. To ensure a single copy or a complete year's supply, fill in the form below, cut it out and send it with your cheque or postal order (made payable to ASP Ltd) to:
Personal Software
Subscriptions,
513 London Road,
Thornton Heath,
Surrey CR4 6AR.

– you can even spread the load with your credit card!

Don't miss out – subscribe now.



SUBSCRIPTION ORDER FORM

Cut out and SEND TO :

Personal SOFTWARE

513 LONDON ROAD,
THORNTON HEATH,
SURREY,
CR4 6AR.

POST FREE SUBSCRIPTION

Please use BLOCK CAPITALS and include post codes.

Name (Mr Mrs Miss)
delete accordingly

Address

Signature

Date

Please commence my subscription to Personal Software with the ... issue.

SUBSCRIPTION RATES (tick ☐ as appropriate)

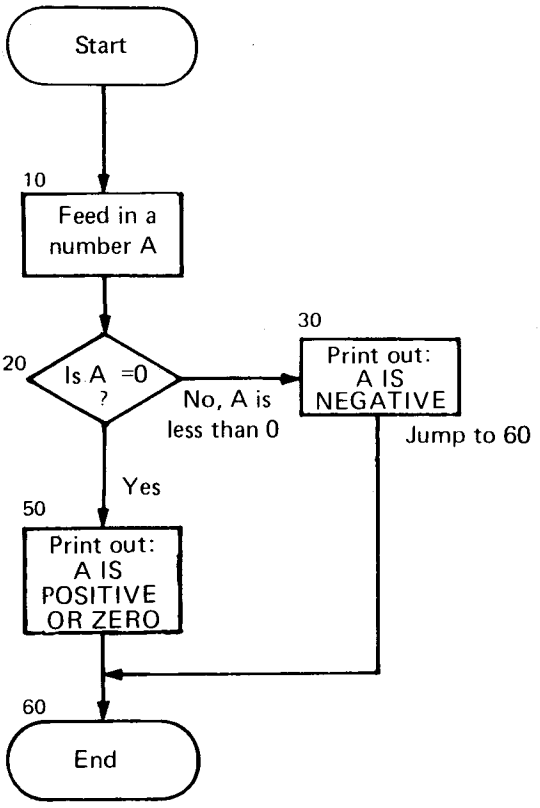
£7.80 for 4 issues UK ☐

£1.95 for a single copy of the ... issue ☐

I am enclosing my (delete as necessary)
Cheque/Postal Order/International Money Order for £
(made payable to ASP Ltd)
OR
Debit my Access/Barclaycard (*delete as necessary)*

Insert card no.

computer. BASIC helps us in this respect, in that a statement is available which allows a flowchart of the type just discussed to be converted for use by the computer, eg:



In BASIC, this has the form:

```
10 INPUT A
20 IF A >= 0 THEN GOTO 50
30 PRINT "A IS NEGATIVE"
40 GOTO 60
50 PRINT"A IS POSITIVE OR ZERO"
60 END
```

Line 20, the IF. . .THEN. . . statement, is of special interest to us. Such a statement is generally built up along the following lines:

Line number IF expression conditional operator THEN statement

In our example, line 20 compares the value of A with 0. If the comparison A>=0 is valid, then there is a jump to line 50 (. . .THEN GOTO 50). If it is invalid, the THEN in the statement will be ignored and the next line to be carried out will be line 30.

Let us now rearrange the program:

```
10 INPUT A
20 IF A<0 THEN GOTO 50
30 PRINT"A IS POSITIVE OR NULL"
40 GOTO 60
50 PRINT"A IS NEGATIVE"
60 END
```

Although we have apparently made a few changes with regard to the first version, a close inspection will reveal that this program is also a conversion of the last short flowchart. Think it over and you'll come to the same conclusion.

Here are further examples of the use of the IF. . .THEN. . . statement:

a) 10 IF A<> B THEN GOTO 30
b) 10 A*B=10 THEN GOTO 30
c) 10 IF A*B <C+D THEN GOTO 30
d) 10 IF (A*A+2*A*B+B*B)=0 THEN GOTO 30
e) In the event of both A and B being unable to take on negative values, the statement:
   10 IF A+B=0 THEN GOTO 30
   is an elegant way of testing whether A and B have the value null simultaneously.

As we can see there are constants, variables, and expressions to the left and right of the relational operators. And it's all legal in BASIC.

Of all the statements we have come across up to now, the IF. . .THEN statement is the most efficient one. It is also called a 'conditional jump', because a jump in the program follows when the condition holds. Fundamentally, on completion of the IF. . . part of the statement, the program can be continued from one of two different points, and for this reason the name 'branching statement' is sometimes met. It *doesn't* hold that a GOTO statement follows IF. . .THEN. . . , and we will see how versatile the IF. . .THEN statement is after completion of the next example.

Program Example 6: Areas of Circles

Due to the importance of the IF. . . statement it should be practised, and for this reason we'll write a short program. Let us ask ourselves the following question: What is the sum of the areas of all the circles with the radius between 1 and 9? The result should be printed so that a blank line appears between the values for 5 and 6.

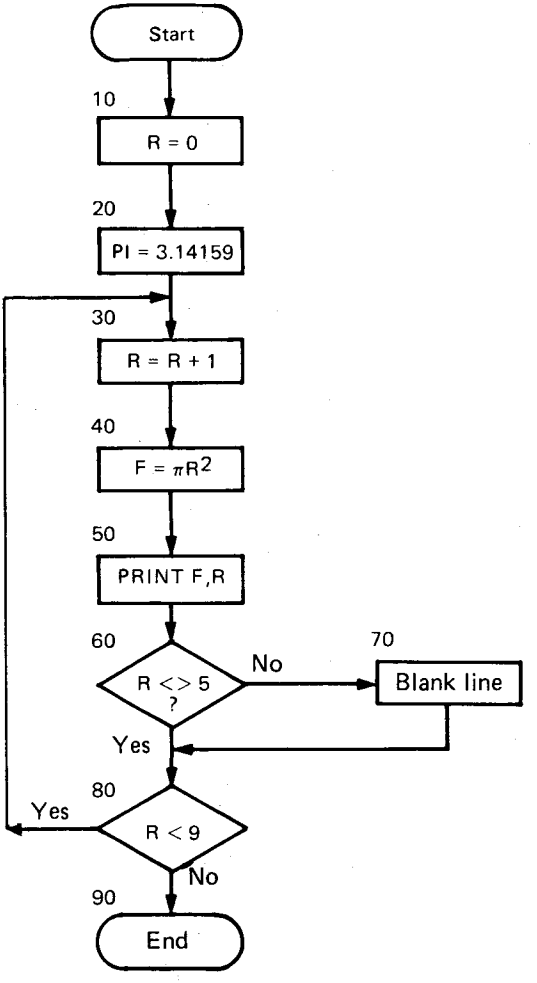
Perhaps a bit of a dull question, but nevertheless good for practice. A solution is shown in the next flowchart.

In line 10 the variable R (radius) is given the value D. R receives the first 'proper' value, namely 1, in line 30. The variable PI on line 20 is assigned a value 3.14159 (a Greek π). PI is needed for the calculation of the area in line 40. This line could also be written:

```
40 F=3.14159*R*R
```

This means, however, that the computer has to translate the constant 3.14159 into a usable form every time line 40 is worked. The number 3.14159 has already been translated and stored under the name PI in line 20: much more efficient.

Line 60 checks whether R is equal to 5. If not, then there is a jump to line 80. In the case of R being equal to 5, line 70, the insertion of a blank line, will be carried out. Finally, in line 80, the program checks to see if the value 9 has been reached. If the condition 'R is less than 9' holds, then the program is continued from line 30 and the radius will be increased by 1. If R<9 doesn't hold, then R has taken the value 9 and the program can be terminated.




```
10 R=0
20 PI=3.14159
30 R=R+1
40 F=PI*R*R
50 PRINT R,F
60 IF R<>5 THEN GOTO 80
70 PRINT
80 IF R<9 THEN GOTO 30
90 END
```

Let us go back to the IF. . .THEN. . . statement, for it is capable of more than was first described. We only mentioned cases where a GOTO statement was also involved. This should not be seen as the only possibility, for THEN can be followed by any complete statement. For example:

```
10 IF A=B THEN LET C=3
```

or simply

```
10 IF A=B THEN C=3
```

The sense of this IF. . .THEN. . . statement still corresponds to our description. If the condition is met, then the statement following THEN will be carried out. If not, THEN will be ignored and the program goes on to the next line. In our example, C=3 for the condition that A=B. We can now write our area program in a more elegant form using this further capacity of the IF. . . statement. From the flowchart we see that a blank line is to appear after the result of R=5. This was achieved by a jump to the PRINT statement. This part of the program can, however, be rearranged:

```
60 IF R=5 THEN PRINT
```

A blank line will be printed for R=5, but not for other values of R. The complete program now reads:

```
10 R=0
20 PI=3.14159
30 R=R+1
40 F=PI*R*R
50 PRINT R,F
60 IF R=5 THEN PRINT
80 IF R<9 THEN GOTO 30
90 END
```

As you would have noticed we can now do without line 70.

It's perhaps easier to understand the IF. . . statement, if one regards it as a construction containing two separate parts. The IF part allows a programmer to pose a question which the computer answers with either 'yes' or 'no'. If the answer is 'yes', then the computer goes on to the THEN part and carries out whatever statement follows. We can even hang on another IF. . .THEN. . . statement! That would, at present, somewhat complicate the matter so we'll content ourselves with another example.

Program Example 7: The Comparison of Two Numbers

```
10 INPUT A,B
20 IF A>B THEN PRINT"A IS GREATER THAN B"
30 IF A=B THEN PRINT"A IS EQUAL TO B"
40 IF A<B THEN PRINT"A IS LESS THAN B"
100 END
```

Line 10 causes the computer to go on stand-by until the programmer has entered two numbers which are then assigned to A and B respectively. Line 20 compares the two

numbers and determines whether A (the number entered first) is greater than B. If this condition is met then the computer prints: A IS GREATER THAN B. If not, the program is continued from line 30 where there is a check for the condition 'A=B'. If this holds, then the computer prints A IS EQUAL TO B, if not, we proceed to line 40. For the case that A is less than B the computer print A IS LESS THAN B. You'd now be right to ask what happens if A is *not* less than B. The answer is simple: The program would go on to line 100. This is logical, because one of the other two conditions handled in lines 20 and 30 has then been met and the text corresponding to the valid condition already printed. A justifiable query could now be raised: If the condition in line 10 is met, the computer prints A IS GREATER THAN B, then one could already consider the program as finished. Unluckily, however, the BASIC Interpreter stubbornly goes on to the next line although it's superfluous. Let's try to bypass the unnecessary program lines:

```
10 INPUT A,B
20 IF A>B THEN PRINT"A IS GREATER THAN B"
25 GOTO 100
30 IF A=B THEN PRINT"A IS EQUAL TO B"
35 GOTO 100
40 IF A<B THEN PRINT"A IS LESS THAN B"
100 END
```

Should the condition in line 20 be valid, then the computer prints A IS GREATER THAN B, proceeds to line 25 and subsequently jumps to line 100. Otherwise, the program moves directly on to line 25 and still jumps to 100! This is not our aim and we have, in fact, made a typical logical mistake, with the result that the computer was not programmed as we intended.

It's an established fact that, in programming, compromises are often necessary. Either the program is kept short and the computer has to do more than is really necessary, or the computer does the minimum, and the program becomes extensive. An example of the latter is shown below in a new formulation of our program.

```
10 INPUT A,B
20 IF A>B THEN 50
30 IF A=B THEN 60
40 PRINT"A IS LESS THAN B"
45 END
50 PRINT"A IS GREATER THAN B"
55 END
60 PRINT"A IS EQUAL TO B"
100 END
```

Instead of five lines we have now got nine, but the computer now doesn't do any unnecessary work. The hobby programmer, when faced with the choice between the two programs, would probably go for the first version, for the simple reason that storage space is normally more important than program execution time. Usually, though, one has to consider each program individually and then decide whether to save storage space or time.

Exercise 3

Write a program which either adds or multiplies two numbers together and then prints the result.

Exercise 4

Write a program which permits the subtotal of continuously entered numbers to be calculated and printed after the entry of each new number. The use of the number 0 should cause the computer to print out the sum up to this point as a 'total', and subsequently start a new summation.

6. One for All

Arrays — DIM — FOR. . .NEXT

Let us suppose that you are a businessman and want to use your computer to calculate the end price for a variety of articles subject to VAT. If the VAT is running at 15% this means that the article price is to be multiplied by 1.15. A short BASIC program could be:

Program Example 8: VAT

(The capability of this program will be increased in the course of the chapter.)

```
NEW
10 INPUT P
20 P=P*1.15
30 PRINT P
RUN
```

Line 10 gives you the opportunity to enter the price of the article, which is then stored by the computer under the name P. In line 20 we then make use of a little trick: The computer is told to multiply the value of P by 1.15 and to store the new number under the storage space reserved for P. We have effectively used the storage space P twice, in the first instance to take up the entry P and secondly to store the 'new' value, which is also called P. In other words the 'old' value is replaced by the 'new' one. This is quite in order as the initial value is only used for the multiplication. This procedure is often adopted to save on variables and consequently storage space. Line 30 is responsible for printing the new value of P.

It would seem troublesome to have to start the program with RUN every time a new value is to be calculated. And with a little alteration it can be avoided:

```
10 INPUT P
20 P=P*1.15
30 PRINT P
40 GOTO 10
```

Line 30 prints the price including VAT. This is followed by a jump back to line 10, and now we can enter the next price for which VAT is to be calculated. This program runs in an endless loop and can only be terminated by depressing the STOP or BREAK key, or by switching off the computer.

Up to now we wouldn't have won any prizes with our computer, in fact a pocket calculator would have done just the same.

Let us now consider a more complicated case: A tradesman buys a definite number of five different articles, for which he then wants to calculate the VAT and the expected returns. A suitable program is:

```
10 INPUT A,S
20 A=A*S*1.15
30 PRINT A
40 INPUT B,S
50 B=B*S*1.15
```

```
60 PRINT B
70 INPUT C,S
80 C=C*S*1.15
90 PRINT C
100 INPUT D,S
110 D=D*S*1.15
120 PRINT D
130 INPUT E,S
140 E=E*S*1.15
150 PRINT E
160 PRINT A+B+C+D+E
```

Line 10 allows for the input of the first article at price A and its quantity. Line 20 calculates the sale price for the article as a whole and stores it again under A. Line 30 permits the sale price to be printed. In line 40 the price of the next article can be entered and this time we use B, for A will be needed again later. It's a different matter though with S. S can be used as often as necessary, because the reserved storage space becomes available as soon as the sale price has been evaluated. This system is repeated until all five articles are entered. Finally line 160 prints out the sum of the individual returns.

Apart from an imposing length, the program hasn't much to offer, and it's worthwhile to think about how we can turn it into a compact and handy package.

If we look at the variables A,B,C,D,E again, then we can conclude that their common factor can be interpreted as a price. The program treats them identically in that they are multiplied with a number and finally summed up. What would happen if the tradesman stocked more articles than could be accounted for with BASIC names?

The possibility of this happening is faint, for BASIC has been equipped to deal elegantly with a large number of variables simultaneously. Instead of the simple variables

A,B,C,D,E,. . .

we can use an alternative nomenclature:

A(1),A(2),A(3),A(4),A(5), . . .
or B(1),B(2),B(3),B(4),B(5), . . .
or C(1),C(2),C(3),C(4),C(5), . . .
etc.

The same name is given to more than one variable (eg, A or B or C), and then the names are numbered, so that one can easily differentiate between them: A(1), A(2), A(3), . . . Each of these numbered variables is completely independent and receives storage space to accommodate its respective value.

This method of designating variables stems from mathematics, where an accepted way of writing is

$$a_1, a_2, a_3, \ldots$$

or

$$x_1, x_2, x_3, \ldots$$

The BASIC language is not provided with small letters or the ability to number these letters in this way, so that one has to improvise: A(1),A(2) etc. These are called 'subscripted variables' with the 'subscript' being the number associated with the variable. Thus A(1) would be read 'A subscript one' or 'A one'.

All the subscripted variables carrying the same name, eg the variables A(1),A(2),A(3), etc., constitute what is called an 'array'.

Let's now go back to our tradesman, and try to write his program in a more elegant form:

```
10 INPUT A(1),S
20 A(1)=A(1)*S*1.15
30 PRINT A(1)
40 INPUT A(2),S
50 A(2)=A(2)*S*1.15
60 PRINT A(2)
    .
    .
    .
160 PRINT A(1)+A(2)+A(3)+A(4)+A(5)
```

One can hardly regard this program as elegant, quite the opposite; we have even made the program longer, in that the simple variable (eg B) has been replaced by a subscripted variable. Nevertheless the program gains clarity as the number of variable names has been reduced. Let's try to simplify the program further:

```
10 I=1
20 INPUT A(I),S
30 A(I)=A(I)*S*1.15
40 PRINT A(I)
50 I=I+1
60 GOTO 20
```

This program is considerably shorter, although we shouldn't overlook the fact that it embraces both a drawback and something new to us. Looking at line 20 we note that the variable A is not subscripted with some number or other but with the letter I. According to previous experience we know that I is the name of a variable. We also know that storage space and a numerical value are assigned to each variable on the condition that the assignment step has been carried out at some earlier stage. In our case this happens in line 10. The variable in line 20 is therefore A(1) and not A(I)! The fact that the subscript (or 'index') can be a variable and not just a whole positive number is new to us. If the BASIC system runs up against such a variable, eg A(I), it first seeks the numerical value of I at this point, inserts it as index in A(I) and then continues with the program.

It may seem a little confusing, but the method does have an advantage. Line 30 calculates A(1) and line 40 causes the result to be printed. In line 50 the value of I is increased by 1 and now becomes 2. Line 60 triggers a jump to line 20, and with I=2 the BASIC system executes the statement

```
20 INPUT A(2),S
```

The lines 30 and 40 which follow are also concerned with A(2). Then the program

progresses for the second time to line 50. The value of I is raised from 2 to 3, and the program is restarted. This repeats itself for ever increasing values of the index I.

The drawback mentioned earlier is found in the fact that the program doesn't know when to stop and consequently runs until the emergency button — the Stop or Break key — is depressed. Additionally, the information over the total expected is missing. We can remedy this by supplementing the program:

```
10 I=1
20 INPUT A(I),S
30 A(I)=A(I)*S*1.15
40 PRINT A(I)
50 I=I+1
60 IF I<=5 THEN GOTO 20
70 PRINT A(1)+A(2)+A(3)+A(4)+A(5)
```

The 'unconditional jump' in line 60 has been replaced by a 'conditional jump':

```
60 IF I<=5 THEN GOTO 20
```

As long as I is less than or equal to 5, the program makes a jump back to line 20, where the next article price can be entered. As soon as the variable I becomes greater than 5 the sum of all returns is printed. Although this is already a compact program, we still have room for improvements.

DIM

If one does decide to use subscripted variables, then a logical assumption is that the BASIC program has a corresponding storage space at its disposal. The computer, however, is not a mindreader and before the first indexed variable appears it has to reserve a whole row of storage spaces for this 'array'.

Consider the following example:

```
10 INPUT K,I
20 A(I)=K
```

The Ith variable in the array A(I) is allocated the value of K. If the user enters the value 1 for I and 12.53 for K, then line 20 becomes A(1)=12.53. On the other hand the user could just as easily choose 100 for I, and line 20 would read A(100)=12.53. As you can see the computer is not secure against the element of surprise.

The BASIC system offers a means of overcoming this. Before an indexed variable is used, one informs the BASIC system of the possible size of the index. One can also speak of the 'dimension' of the array. Should the programmer wish to use the array A(I) containing 20 variables, then the following line has to appear in the program:

```
DIM A(20)
```

The additional choice of a further array of 30 under the name B(I) could look like:

```
DIM A(20),B(30)
```